

## Assignment 3

### Linux Kernel Module - Page Table Walker

#### Objective

The purpose of this assignment is to gain experience writing code that executes in kernel (protected) mode in Linux by implementing a Linux Kernel Module. It is best to complete this assignment on a virtual machine. In case of problems, it is easier to restart a virtual machine, than to restart the host system.

The goal of the assignment is to build a kernel module that traverses the list of running processes and generates report that displays the process ID, process name of all processes with PID > 650, and the total number of pages allocated in memory. This report output from the kernel module should be made to the kernel log files (/var/log/syslog in Ubuntu) and to a Linux “proc” file under the “/proc” directory.

The **extra credit** goes to the extra information in the report that show the number of pages that are allocated contiguously vs. non-contiguously in physical memory for each process.

#### Sample Output

Output should be generated in comma separated value (CSV) format. The columns “contig\_pages” and “noncontig\_pages” are **optional**.

```
PROCESS REPORT:
proc_id,proc_name,contig_pages,noncontig_pages,total_pages
654,auditd,95,353,448
663,audispd,61,180,241
665,sedispach,54,202,256
679,rsyslogd,545,443,988
680,smartd,177,406,583
. . .
5626,kworker/0:2,315,500,815
5641,kworker/3:0,315,500,815
6165,sleep,34,121,155
6182,cats,32,120,152
TOTALS,,218407,124327,342734
```

The first line says “**PROCESS REPORT:**”. The second line is a comma-separated header line describing the columns. The columns are **proc\_id** for the process ID, **proc\_name** for the process name (the comm field of task\_struct), and **total\_pages** which is the total number of memory pages allocated (i.e., in use) for the process.

**Optional** columns include: **contig\_pages** which is the number of contiguous pages, and **noncontig\_pages** which is the number of non-contiguous pages.

The last line of the report is **optional**. The last line displays **TOTALS** in the first cell, a blank value in the second cell and provide a sum of the total number of contiguous pages, total number of non-contiguous pages, and the total number of pages for all processes with PID > 650.

## Instructions

For walking the page table, it should be possible to export the CSV output to a spreadsheet to support further analysis of the data. (NOTE: it is non-trivial to divide to calculate averages in kernel mode.) For example, it is easy to calculate the average continuous pages per process for PIDS>650 after generating the report in kernel.

A sample kernel module “hello\_module.tar.gz” can be downloaded from folder “A3: Page Table Walker” in Canvas.

To extract the sample kernel module:

```
tar xzf hello_module.tar.gz
```

To build the sample module:

```
cd hello_module/  
make
```

To remove a previously installed the module:

```
sudo rmmod ./helloModule.ko
```

To install a newly built module:

```
sudo insmod ./helloModule.ko
```

This sample kernel module prints messages to the kernel logs.

The “dmesg” command provides a command to interface with kernel log messages, but it is simple enough to just trace the output using:

```
sudo tail -fn 50 /var/log/messages
```

NOTE: Your Ubuntu may not have /var/log/messages anymore and you can find the same information from /var/log/syslog

**\*\*\*YOUR KERNEL MODULE SHOULD BE RENAMED TO “procReport”\*\*\***

**FAILURE TO RENAME THE MODULE WILL RESULT IN A 10 POINTS DEDUCTION**

Your kernel module should produce output as in the example described above. To support development, it may be helpful to begin by writing code that sends output to the system log files using printk, and then later, go back and refactor to send output to the proc file.

Here are some references describing how to create the proc file kernel module interface:

<https://linux.die.net/lkmpg/>

<https://linux.die.net/lkmpg/x769.html>

<http://tuxthink.blogspot.com/2013/10/creating-read-write-proc-entry-in.html>

<https://stackoverflow.com/questions/8516021/proc-create-example-for-kernel-module/> (Hint: Useful proc file example)

Robert Oliver provides a helpful how-to Linux kernel module using Ubuntu 16.04 here:

<https://blog.sourcerer.io/writing-a-simple-linux-kernel-module-d9dc3762c234>

A good starting point is to first iterate the set of processes in Linux, and print out the proc ID and name. Please name your “/proc” file as “**proc\_report**” in the /proc directory.

## Walking the Page Tables

An excellent starting point is provided from this stack overflow post regarding performing virtual to physical page address translations. But this example is only for 4-levels of Linux paging:

<https://stackoverflow.com/questions/20868921/traversing-all-the-physical-pages-of-a-process>

The code to walk the memory page tables is as follows: Each Linux process has a field in task\_struct called “mm” which means “memory map”. The memory map contains a list of contiguous blocks of virtual memory addresses (vma). Each virtual memory block has a start and an end address demarking the contiguous set of virtual memory pages. For each block, the total number of pages can be calculated using PAGE\_SIZE. Thereafter, the total number of pages allocated (i.e., in use) of the process can be calculated by summing over all blocks. The following code is used to translate from virtual address to physical address. However, the similar logic can be used to calculate the total number of pages.

```
struct vm_area_struct *vma = 0;
unsigned long vpage;
if (task->mm && task->mm->mmap)
    for (vma = task->mm->mmap; vma; vma = vma->vm_next)
        for (vpage = vma->vm_start; vpage < vma->vm_end; vpage += PAGE_SIZE)
            unsigned long physical_page_addr = virt2phys(task->mm, vpage);
```

## OPTIONAL: Virtual to Physical Translation

We walk these virtual pages and ask our virtual to physical address translation function (virt2phys) to translate our virtual address to a physical address. Once the physical address is known, it is possible to determine, for each process, how many of the pages are adjacent in physical RAM.

Here is code to obtain the physical address of a memory page.

```
struct vm_area_struct *vma = 0;
unsigned long vpage;
if (task->mm && task->mm->mmap)
    for (vma = task->mm->mmap; vma; vma = vma->vm_next)
        for (vpage = vma->vm_start; vpage < vma->vm_end; vpage += PAGE_SIZE)
            unsigned long physical_page_addr = virt2phys(task->mm, vpage);
```

The function virt2phys() must be implemented based on the following pseudo code (i.e., next page). Note if the virtual page is unmapped, the \*\_none() function returns 0 and the virtual page can be ignored.

Essentially, the virtual memory areas of a process are walked. VMAs are described by the struct `vm_area_struct`. A linked list is provided which can be walked to obtain the virtual addresses.

**For `virt2phys`, implement a function based on the code that takes in a `task->mm` and a `vpage` to then return an unsigned long address.**

**For this assignment analyze only PIDs > 650.**

If `virt2phys` should return 0 at one of the stages, for example while translating either the `pgd`, `pud`, `pmd`, or `pte`, it is ok for this assignment to ignore the 0, and just keep looping. Treat a 0 as an unmapped/untranslatable page entry for this assignment.

Linux provides structures for a 5-level page table where `pgd_t` is the highest-level page directory, `p4d_t` is the fourth-level page directory, `pud_t` is the upper page directory, `pmd_t` is the middle page directory, and `pte_t` is a page table entry. There is no guarantee that all of these 5-levels will be physically backed

```
//...
//Where virt2phys would look like this:
//...

pgd_t *pgd;
p4d_t *p4d;
pud_t *pud;
pmd_t *pmd;
pte_t *pte;
struct page *page;
pgd = pgd_offset(mm, vpage);
if (pgd_none(*pgd) || pgd_bad(*pgd))
    return 0;
p4d = p4d_offset(pgd, vpage);
if (p4d_none(*p4d) || p4d_bad(*p4d))
    return 0;
pud = pud_offset(p4d, vpage);
if (pud_none(*pud) || pud_bad(*pud))
    return 0;
pmd = pmd_offset(pud, vpage);
if (pmd_none(*pmd) || pmd_bad(*pmd))
    return 0;
if (!(pte = pte_offset_map(pmd, vpage)))
    return 0;
if (!(page = pte_page(*pte)))
    return 0;
physical_page_addr = page_to_phys(page);
pte_unmap(pte);
// handle unmapped page
if (physical_page_addr==70368744173568)
    return 0;
return physical_page_addr;
```

by all HW (CPUs) or all specific compilations of the Linux kernel. But Ubuntu on VirtualBox with a kernel >= 4.11 should work.

- `pgd_t` is the page directory type (5th level)
- `p4d_t` is the page directory type (4th level)
- `pud_t` is the page upper directory type (3rd level)
- `pmd_t` is the page middle directory type (2nd level)

- `pte_t` is the page table entry type (1st level)
- `pgd_offset()`: returns pointer to the PGD (page directory) entry of an address, given a pointer to the specified `mm_struct`
- `p4d_offset()`: returns pointer to the P4D (level 4 page directory) entry of an address, given a pointer to the specified `mm_struct`
- `pud_offset()`: returns pointer to the PUD entry (upper pg directory) entry of an address, given a pointer to the specified PGD entry.
- `pmd_offset()`: returns pointer to the PMD entry (middle pg directory) entry of an address, given a pointer to the specified PUD entry.
- `pte_page()`: pointer to the struct `page()` entry corresponding to a PTE (page table entry)
- `pte_offset_map()`: Yields an address of the entry in the page table that corresponds to the provided PMD entry. Establishes a temporary kernel mapping which is released using `pte_unmap()`.

Reference slides describing Linux virtual memory areas are here:

<http://www.cs.columbia.edu/~krj/os/lectures/L17-LinuxPaging.pdf>

For determining contiguous page mappings, just calculate the next address by adding `PAGE_SIZE` to the current page address. If the next page in the process's virtual memory space is mapped to the current page's physical location plus `PAGE_SIZE` then this is considered a contiguous page – record a “tick” for a contiguous mapping. IF not, record a tick for a “non-contiguous” mapping.

## What to Submit

For this assignment, submit a tar gzip archive as a single file upload to Canvas.

Tar archive files can be created by going back one directory from the kernel module code with “`cd ..`”, then issue the command “`tar czf hello_module.tar.gz hello_module`”. Name the file the same as the directory where the kernel module was developed but with “.tar.gz” appended at the end: `tar czf <module_dir>.tar.gz <module_dir>`.

**Please rename modules to something other than `hello_module`.** To rename a directory in Linux use: “`mv hello_module my_proc_module`”.

## Pair Programming (optional)

*Optionally*, this programming assignment can be completed with two (at most) person teams.

If choosing to work in pairs, **only one** person should submit the team's tar gzip archive to Canvas.

Additionally, **EACH** member of a pair programming team must provide an **effort report** of team members to quantify team contributions for the overall project. **Effort reports** must be submitted INDEPENDENTLY and in confidence (i.e. not shared) by each team member to capture each person's overall view of the teamwork and outcome of the programming assignment. Effort reports are not used to directly numerically weight assignment grades.

**Effort reports** should be submitted in confidence to Canvas as a PDF file named: "effort\_report.pdf". Google Docs and recent versions of MS Word provide the ability to save or export a document in PDF format.

Distribute 100 points for the categories to reflect each teammate's contribution for: research, design, coding, testing. Effort scores should add up to 100 for each category. Even effort 50%-50% is reported as 50 and 50. **Please do not submit 50-50 scores for all categories. This is not necessarily realistic or possible.** Ratings should reflect an honest confidential assessment of team member contributions. ***50-50 ratings and non-confidential scorings run the risk of an honor code violation.***

Here is an **effort report** for a pair programming team (written from the point of view of Jane Smith):

1. John Doe: Research 24, Design 33, Coding 71, Testing 29
2. Jane Smith: Research 76, Design 67, Coding 29, Testing 71

Team members may not share their **effort reports**, but should submit them confidentially in Canvas as a PDF file. Failure of one or both members to submit the **effort report** will result in both members receiving NO GRADE on the assignment... (*considered late until both are submitted*)

#### Disclaimer regarding pair programming:

The purpose of TCSS 422 is for everyone to gain experience programming in C while working with operating system and parallel coding. Pair programming is provided as an opportunity to harness teamwork to tackle programming challenges. But this does not mean that teams consist of one champion programmer, and a second observer simply watching the champion! The tasks and challenges should be shared as equally as possible.

## Grading

**NOTE: it is important to decouple the output routines from the report generation. It means that you need to store required information in a data structure at first. Then, printf or write to /proc. Otherwise, a subtle time difference can make the output different.**

This assignment will be scored out of 95 points.

#### Rubric:

110 possible points: (15 extra credit points are available)

Report Total: 70 points

---

15 points	Output of the PID for processes with PID > 650
15 points	Output of the process name for processes with PID > 650
25 points	Output of the number of total pages for PIDs
5 points	<b>OPTIONAL:</b> Output of the number of contiguous pages for PIDs
5 points	<b>OPTIONAL:</b> Output of the number of non-contiguous pages for PIDs
5 points	<b>OPTIONAL:</b> Output the total: # of contiguous pages, # of non-contiguous pages, and # of pages for all processes with PID > 650

Output Total: 20 points

---

- |           |   |
|-----------|---|
| 10 points | Report output is to a Linux proc file named /proc/proc_report<br><i>&gt;&gt;&gt; 5 points - decoupling output routines from report generation</i> |
| 10 points | Report output is sent to the kernel log file<br><i>&gt;&gt;&gt; 5 points - decoupling output routines from report generation</i>                  |

Miscellaneous: 20 points

---

- |          |   |
|----------|---|
| 5 points | Kernel module builds, installs, uninstalls                                |
| 5 points | Following the Output requirements as described (even with missing output) |
| 5 points | Kernel module does not crash computer                                     |
| 5 points | Coding style, formatting, and comments                                    |

WARNING!

---

- |           |  |
|-----------|--|
| 10 points | Automatic deduction if the module is not called "procReport" |
|-----------|--|